

Towards Refactoring the DMF to Support Jini and JMS DMS in GIPSY

Yi Ji, Serguei A. Mokhov, and Joey Paquet
 Computer Science and Software Engineering
 Concordia University, Montreal, QC, Canada
 {ji_yi,mokhov,paquet}@cse.concordia.ca

Abstract

In this paper we report on our re-engineering effort to refactor and unify two somewhat disjoint Java distributed middleware technologies – Jini and JMS – used in the implementation of the Demand Migration System (DMS). In doing so, we refactor their parent Demand Migration Framework (DMF), within the General Intensional Programming System (GIPSY). The complex Java-based GIPSY project is used to investigate on the intensional and hybrid programming paradigms.

1 Introduction

The GIPSY research prototype system [26] (see Section 2) is a collection of replaceable Java components arranged primarily into three main packages – the collection of compilers for the programming languages of interest (GIPC – core – Lucid-based intensional dialects [3], and hybrid dialects, primarily mixing Lucid code and Java code to various degrees), runtime programming environment (RIPE) (currently a loose set of user interaction components), and the run-time systems – the general education engine (GEE). We focus on some aspects of the latter in this paper, specifically its distributed subcomponents that implement two instances of the demand-migration system (DMS) using two distributed Java middleware technologies, namely Jini and JMS for comparative studies of evaluation of hybrid programs to gain insight on the technologies and various their parameters from programmability to scalability metrics among others. Here we report some of our findings through development and experiments.

Problem Statement One of the main reasons necessitating this study is the hybrid intensional programming aspect the the GIPSY platform is there to investigate among other things. Lucid programs are naturally parallel and expressive [3, 31] as well as context-oriented with contexts as first class values [32, 27]. Yet, in itself Lucid is rather simple functional language for computation, and does not have rich I/O and other support, so it should rely on the existing libraries and frameworks when such needs arise leading the way to hybrid programming paradigms involving a Lucid dialect and Java in our case. Earlier (while still very valuable to the community, but relatively non-scalable and umaintainable) solutions, were proposed and their enhancement with hybridification of Lucid and C [9, 10] or later C++ [19]. Since GIPSY’s architecture was first proposed and evolved in the past 10 years to be extendable and component-based, hybrid prototype dialects emerged combining Java and Lucid – JLucid (Lucid program primarily calls only Java methods), Objective Lucid (Lucid to access to object properties of Java objects) [15] and later JOOIP (Java-based OO Intensional Programming) language [33] that enabled bidirectional Lucid being able to access Java members and, at the same time, Java classes could contain fragments written in Lucid in them. (Further work in programs involves extension of these in the form of Forensic Lucid [17] and MARFL [16]). To support the evaluation of programs written in these hybrid dialects, the runtime (GEE) of GIPSY has to scale to be able

not only to locally compute light-weight Lucid fragments locally, but also compute the hybrid “heavy” Java fragments – given the latter can take a lot of computation and I/O resources, and natural parallelization of Lucid, the hybrid components are proposed to be evaluated distributively. Then the problem becomes which distributed middleware technologies to pick. As a proof of concept, the DMF was defined in Java and was implemented using two different Java middleware technologies, Jini (by Vassev et al. [29]) and JMS (Pourteymour [22]). But those two were development in the simulated prototype environment, relatively isolated from the core GIPSY project and from each other.

Proposed Solution To enable consistent comparative studies of Jini and JMS in the GIPSY multi-tier environment and hybrid language paradigms [21, 18, 7] for points of scalability [11], usability, programmability, deployment, and other aspects, we unify the two Java implementations of Jini and JMS DMS under GEE and its multi-tier architecture and redefine the practical meaning of certain of its components and do extensive testing of both. Hereafter, we report on our experience in this regard.

Organization We proved the necessary background of the Java-based GIPSY project and its distributed middleware technologies used in Section 2. We then describe the objectives of this work in Section 3.1 and present the methodology and the approach in Section 3, and finally we conclude in Section 5.

2 Background

The General Intensional Programming System (GIPSY) provides a platform to investigate the possibilities of the intensional programming [20]. The intensional programming model, in the sense of Lucid, is a declarative and functional programming language paradigm where the identifiers are evaluated in multidimensional context spaces. The GIPSY compiler translates any flavor of intensional program into a source-language independent GIPL program, and the GIPSY runtime system executes the GIPL program using an evaluation model called education. In the demand-driven education model, an initial demand requesting the value of a certain identifier is generated, and to consume this demand, new demands are generated to request the values of the identifiers constituting the expression defining the initial identifier, and similarly these demands further generate new demands until eventually some of the demands are evaluated and propagated back in the chain of demands, so that the identifiers whose value depend on them can be evaluated in turn, and eventually the initial identifier is evaluated [20]. This demand-driven education model naturally supports distributed execution of intensional programs [10, 19].

The Demand Migration Framework (DMF) for the GIPSY runtime system was proposed for the distributed and demand-driven execution of intensional programs by Vassev et al. [26, 30, 29, 28, 22, 15], and two principle Demand Migration Systems (DMS) as of this writing – based on Jini and JMS Java middleware technologies were provided to implement the DMF [23, 24]. The basic idea of the DMF is to provide a generic framework defining interface to migrate/propagate demands among distributed demand generators and demand workers, where the generators generate demands and the workers compute the demands.

A follow up work by Ji et al. [11] further streamlined the code for unification while performing scalability studies for the the Jini and JMS implementations of DMS in GIPSY (see Section 4).

We briefly introduce the Jini DMS and the JMS DMS architectures in the sections that follow.

2.1 Jini DMS

Jini is a Java-based and service-oriented middleware technology for building distributed systems consisting of Jini services and clients [12, 1, 6]. Now officially known as *Apache River* [2], Jini defines a set of specifications and provides implementation of several basic services such as lookup discovery, leasing and transaction services. It also provides a Java implementation of the Tuple Space called the JavaSpace service that enables distributed Jini clients to read, write and remove serialized Java objects stored in the shared object repository. JavaSpace supports object persistence so that when restarted, the objects stored in the JavaSpace can be recovered.

The Jini DMS of the GIPSY runtime system was implemented by Vassev with his proposal of the first Demand Migration Framework (DMF) [29, 28]. As shown in Figure 1, the Jini DMS consists of Demand Generators Demand Workers, Jini Transport Agent (Jini TA) and JavaSpace. The Demand Generators and Workers communicate among each other by sending and reading *demands* into and from the JavaSpace with the aid of the Jini TA. A *demand* is a serialized Java object containing information for the evaluation of Lucid identifiers that require functional computation. A typical demand-migration process is as follows:

1. when parsing the abstract syntax tree of a hybrid Lucid program, the Demand Generators encounter identifiers whose values depend on certain functional computations, so the Generators generate *pending* demands to request these computation and use the TA to send those *pending* demands into the JavaSpace;
2. the Demand Workers then use the TA to pick up the *pending* demands from the JavaSpace and carry out the functional computations requested, and send the *computed* demands back to the JavaSpace;
3. the Demand Generators then use the TA to pick up the *computed* demands from the JavaSpace and retrieve the values of the identifiers.

The Jini TA in Figure 1 consists of Jini TA proxies and a Jini TA backend. Each Demand Generator or Worker can obtain a Jini TA proxy object using the Jini lookup discovery service, and uses this TA proxy to communicate with the remote Jini TA backend via remote method invocation. To write a demand into the JavaSpace, once invoked remotely by the Demand Generator or Worker, the Jini TA backend uses a local Demand Dispatcher object to wrap the demands as JavaSpace entries and send the entries into the remote JavaSpace. The reference to the remote JavaSpace service held by the Demand Dispatcher is also looked up via the Jini lookup discovery service. Similarly, once invoked remotely to read a demand from the JavaSpace, the Jini TA backend uses the Demand Dispatcher object to retrieve the corresponding JavaSpace entry and unwrap it, and returns the demand to the remote Demand Generator or Worker.

The separation of the Jini TA into proxies and the remote backend it allows diverse Jini TA implementations to be integrated into the system without affecting existing components, and may increase system availability by allowing the Demand Generators and Workers to connect to any TA registered in the Jini lookup discovery service. However, its disadvantages are that the additional remote method invocation is redundant that it increases communication cost and undermines performance.

2.2 JMS DMS

JMS is short for Java Message Service that is a Java-based and message-oriented middleware technology [4, 13, 8]. It defines a set of API specifications and has several implementations such

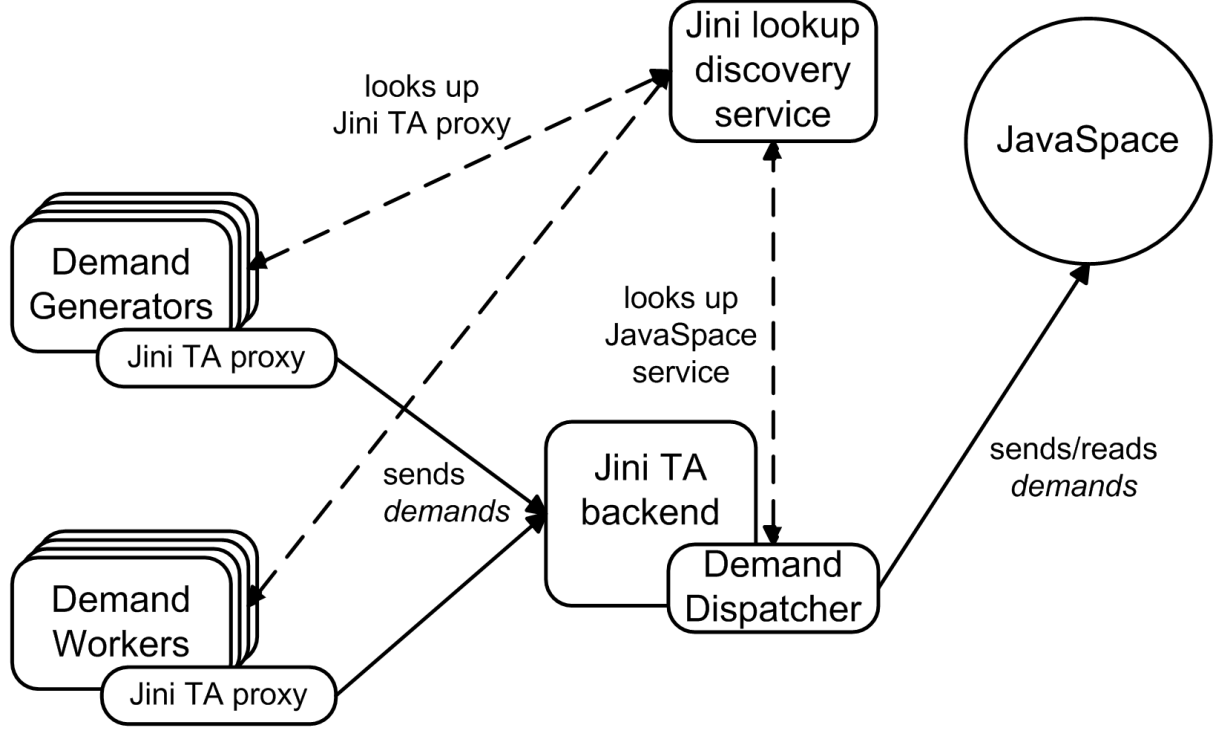


Figure 1: Jini DMS architecture

Open Message Queue and JBoss Messaging. Basically a JMS implementation is a JMS broker providing messaging services to its clients. The JMS clients fall into two domains: message producer and consumer, and message publisher and subscriber. The producer/consumer domain is for end-to-end messaging, meaning that each message is sent by one producer, stored in the message queue in the broker, and received by only one consumer; whereas the publisher/subscriber is for broadcast messaging, in which each message is sent by one publisher, but can be received by multiple subscribers. Compared to Jini JavaSpace, JMS broker provides additional services to ensure availability and reliability such as delicate memory management, flow control, various acknowledgment models and better polished transactions. JMS also supports message persistence.

The JMS DMS was first implemented by Pourteymour [22, 23, 24] based on the message producer/consumer model. The alternative publisher/subscriber model was not adopted because it does not allow newly registered subscribers to receive messages that were published before their registration, whereas the GIPSY must allow workers to pick up *pending* demands in the form of messages sent at anytime. As shown in Figure 2, the JMS DMS consists of Demand Generators, Demand Workers, JMS TAs and the JMS broker service. Similar to the Jini DMS, the Demand Generators generate demands and the Workers compute the demands, and the demands are migrated among them via the TAs and the JMS broker service. However, in this JMS DMS, to send a demand passed by the Demand Generator or Worker, the JMS TAs wrap the demands into object messages and send them directly into the message queues managed by the broker; to read a demand, the JMS TAs directly read object messages from the message queues, unwrap and return the demands to the Demand Generator or Worker. Compared to the Jini DMS, the JMS DMS has a simpler TA architecture without unnecessary communication

cost, and it does not have a concrete Demand Dispatcher to wrap around the message queue since the JMS API and broker was reckoned as the role of the Demand Dispatcher. However, since the JMS TA in the JMS DMS is directly instantiated by the Demand Generator or Worker, once the TA implementation is changed, the Java source code of the existing components will be affected, which is inconvenient to add new TA implementations that are based on different middleware technologies.

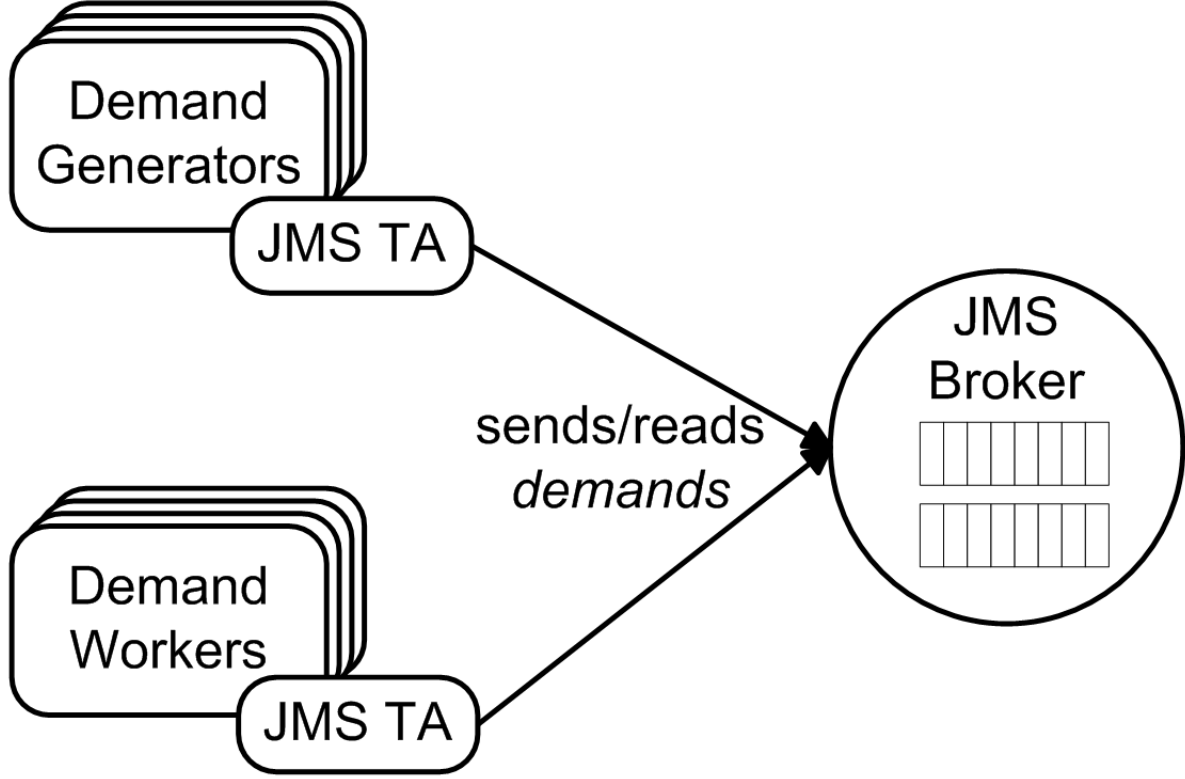


Figure 2: JMS DMS architecture

3 Methodology

We studied the original implementations of Jini and JMS DMS to be able to run both consistently or even concurrently within one GIPSY computing network instance. With this preliminary study we defined a number of required objectives related to the actual refactoring, as well as identifying which technology is more appropriate than the other and under which condition in the consistent uniform setup; including common interfaces, glue, data structures and the like.

3.1 Objectives

- Make JMS [25] and Jini [12] look similar, i.e. be a part of the same interchangeable framework's implementation.
- Redefine the roles of Demand Dispatcher and the Transport Agent (see Figure 3) for the dispatcher to be more of decision maker and a scheduler for the generators, etc. rather

than being attached to the demand store.

- Compare Jini and JMS vs JVM performance and scalability of computation.
- Compare programmability of the two APIs.
- Compare ease of deployment and startup of JMS and Jini from the same common point.

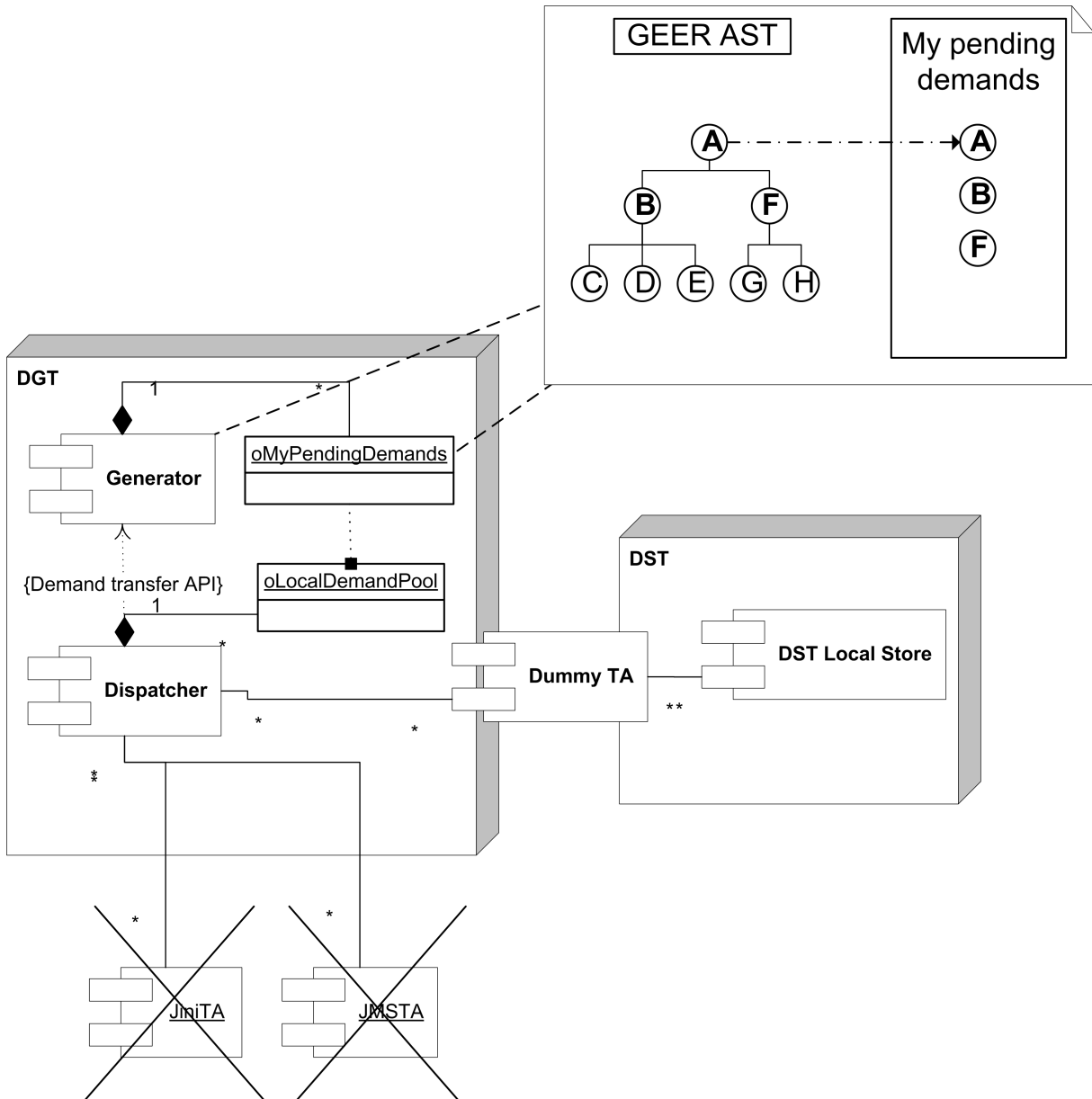


Figure 3: Demand Generator and Demand Dispatcher relationship

3.2 Making Jini and JMS similar

Before the refactoring process, the class diagram of the Jini and JMS DMSs is shown in Figure 4. In this class diagram, the Jini Demand Dispatcher communicates directly with the JavaSpace,

and is used by the JTABackend that is remotely invoked by the JINITransportAgentProxy. The JMSTA communicates with Message Queue directly and had no DemandDispatcher. The JINITransportAgentProxy and the JMSTA inherit different interfaces as they expose too much middleware-dependent features, such as the UUID in Jini and the connection setup phase in JMS, and therefore each DMS has their own Demand Generator and Worker since the Generator and Worker instantiate their TAs directly and use them in a middleware-dependent way.

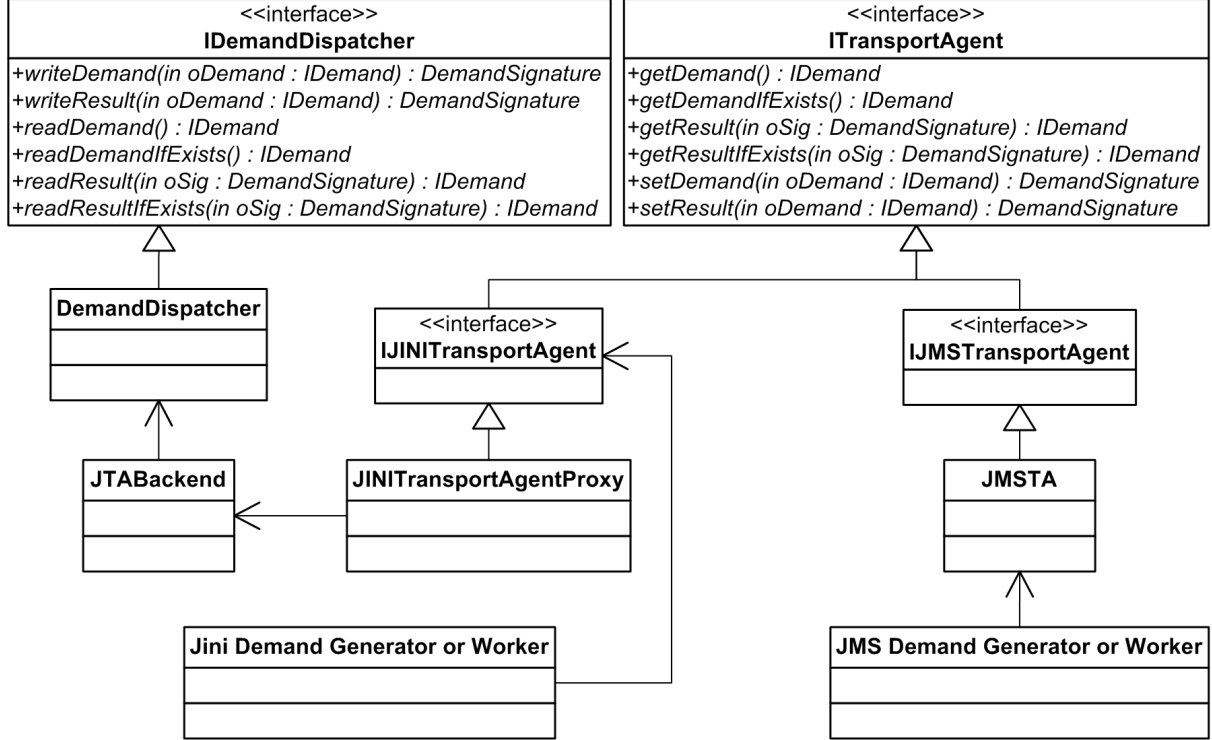


Figure 4: DMS class diagram before refactoring

The refactoring began with creating a new class called JiniTA, moving the original business logic of the DemandDispatcher into this JiniTA, and updating all the references to the original DemandDispatcher by pointing them to the new JiniTA. Then we encapsulated all the middleware-dependent logic, such as JMS connection setup and teardown, inside each TA implementation, and made them directly inherit the ITransportAgent interface. Having done so, we removed all the TA-implementation-dependent logic, such as TA instantiation, from the Demand Generators or Workers, so that they only reference ITransportAgent only. Then we used DemandDispatcher to delegate ITransportAgent, and replaced the usage of ITransportAgent with IDemandDispatcher inside the Demand Generator and Worker, so that the Demand Generators or Workers now talk to IDemandDispatcher only and in the future we could easily add scheduling logic into the DemandDispatcher to decide when, where and using what TA to send or receive demands. The class diagram of the DMS after our refactoring is shown in Figure 5. We also unified the demand classes we use and separate them into subclasses due to their different purposes, such as procedural identifier evaluation, intensional identifier evaluation, runtime-resource acquisition and system management, as shown in Figure 6.

To ease the addition and switching of TA or Demand Dispatcher implementations, we use Java Reflection to instantiate each TA or Demand Dispatcher implementation by the name,

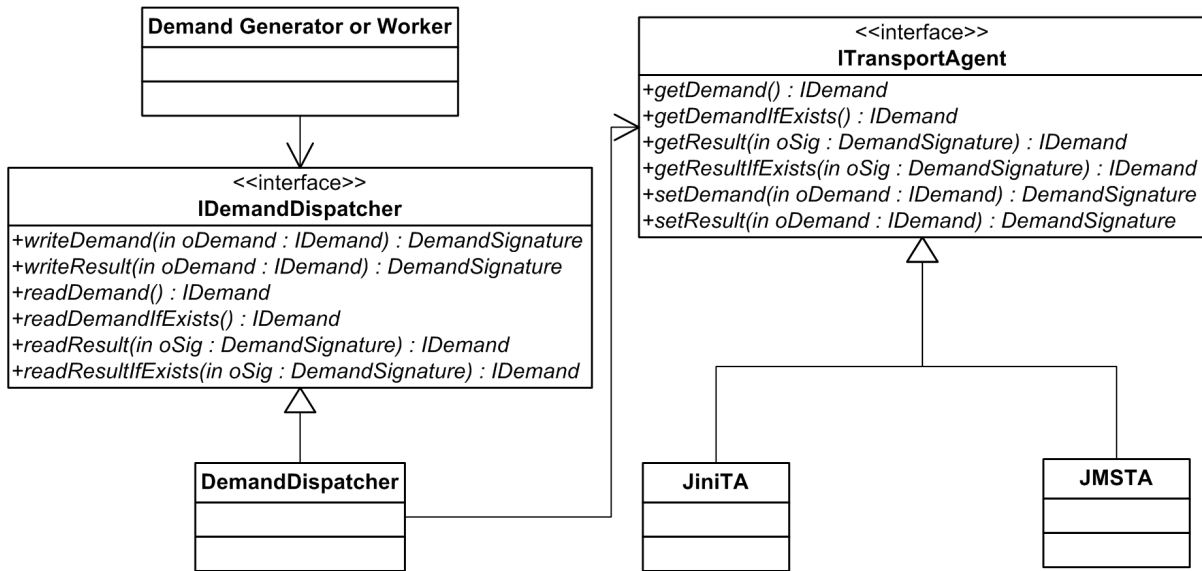


Figure 5: DMS class diagram after refactoring

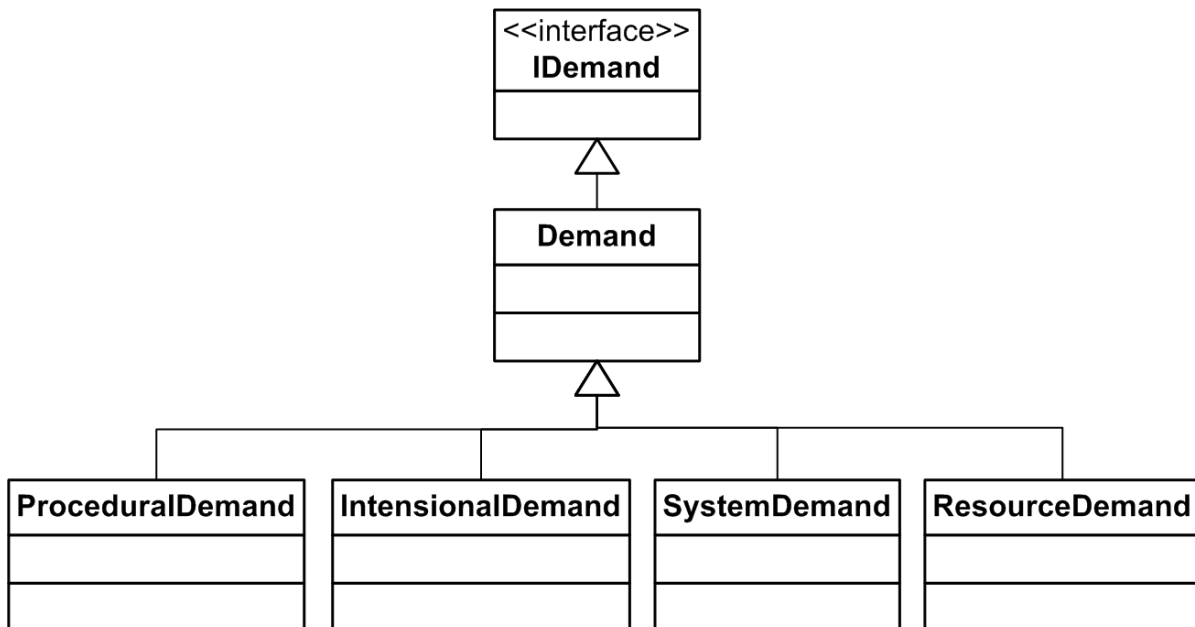


Figure 6: class diagram of demands

where the name of the implementation is a character string passed via network or stored in files. In this way the two DMSs are interchangeable without changing the Java source code of the Demand Generator and the Demand Worker, so that we can use the same set of Demand Generators and Demand Workers to a comparative study of the two DMSs. With proper testing and upon approval from all group members, we committed the changes and cleaned up the affected code remained.

4 Results

We compared the following three aspects of the Jini and JMS DMSs based on our refactoring work.

4.1 Ease of programming

In our experience, the programming of Jini application requires the knowledge of several separate but also correlated Jini concepts and services, such as service lookup discovery, leasing, JavaSpace and transaction. Such knowledge requires time and effort to collect, study and put into practice, especially if additional quality of service (QoS) is required. For example, Jini does not provide additional memory management besides the memory tuning available to the Java Virtual Machine (JVM), therefore if stronger memory management mechanism is required, programmers will have to code their own Jini extension to enhance memory management. In contrast, JMS is a mainstream middleware technology and has relatively integrated services, more QoS choices, well managed documentation and easily understandable tutorials, which is easier for programmers to learn and put into practice.

4.2 Ease of deployment

Jini requires only a set of .jar and configuration files to start its services, therefore has a light weight (the size of all .jar files is less than 4 MB) and can be easily deployed across different platforms, as long as Java is installed in the machines. In contrast, JMS requires platform-dependent executable binary files, such as .exe files in Windows, to start and manage the broker service, therefore it has a larger size (in the case of 32-bit Windows version, the size of all executable files and .jar files is around 20 MB), and requires those platform dependent executable files to be deployed across different platform.

4.3 Runtime issues

We tested that when the Jini DMSs was storing increasing amount of demands, the Jini DMS would run out of memory and crash in the end, even if the data persistence feature was turned on. This shows the storage capacity of the Jini JavaSpace is constrained by its memory, and it provides no mechanism to prevent JVM crash. In contrast, the JMS DMS with message persistence turned on can swap persistent messages into its persistent storage, such as files, to reduce its memory usage once the memory usage exceeds certain threshold. Therefore when the GIPSY runtime system is facing increasing amount of demands, the JMS DMS with message persistence is a better choice when the system availability is the dominant concern.

However, when comparing system performance in the sense of throughput of concurrent Pi calculation demands when the demands were generated by a multi-threaded Generator and computed by multiple Demand Workers distributed in different computers with maximumly

two Workers per computer, we found that as the number Workers increases, the Jini DMS provided a higher throughput than the JMS DMS, and the JMS DMS reached its throughput saturation when there were 10 Workers deployed. The test result is shown in Figure 7, and the test was performed in a lab with computers with the same hardware and operating system environment shown in Figure 1. These computers and their corresponding switch ports have 100 Mbps maximum speeds, with each machine connected to one switch port and all of them on the same subnet and VLAN. This test shows that in the sense of throughput, the Jini DMS is a better choice than the JMS DMS. We also found that for each Demand Generator or Worker connection, the Jini DMS uses one thread to handle each connection, whereas the JMS DMS uses two threads.

All the above differences show that when deployed in managed network, Jini is more suitable for DMS requiring high throughput but low memory storage and low reliability, whereas JMS is suitable for DMS requiring high reliability and availability.

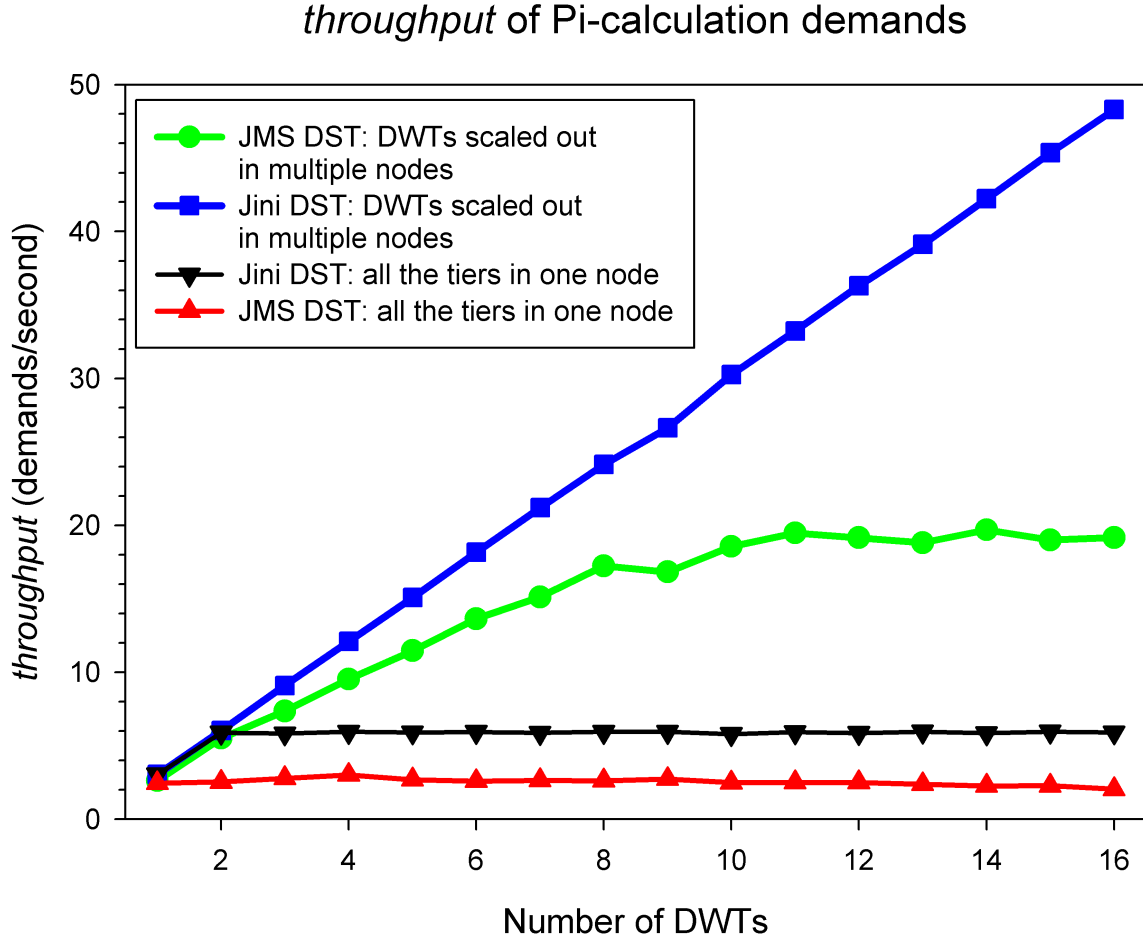


Figure 7: throughput of Pi calculations of the two DMSs

Table 1: Hardware and operating system environment

OS Name	Microsoft Windows 7 Enterprise
Version	Version 6.1.7600 Build 7600
System Type	X86-based PC
Processor	Intel(R) Core(TM)2 CPU 6300 @ 1.86GHz, 1862 MHz, 2 Core(s), 2 Logical Processor(s)
Installed RAM	2.00 GB
Total RAM	2.00 GB
Available RAM	1.06 GB
Total Virtual Memory	4.00 GB
Available Virtual Memory	2.58 GB
Page File Space	2.00 GB

5 Conclusion

We have successfully did the POC integration of the two middleware technologies implementations based on Jini and JMS available to the GIPSY run-time system. In the future work we plan to continue refactoring and cleaning up the other technologies within GIPSY to work together in unison. It is evident, that Jini appears to be easier to work with for development and deployment, but its memory-bound scalability is more problematic than that of JMS, so JMS-based implementation is general more reliable, but Jini DMS offers higher throughput over JMS. For in-depth results of the initial study on various scalability metrics please refer to [11]. Some significant redesign was also necessary to make the two implementations work together consistently, with a potential payoff any new, better or worse, implementations for comparative studies like we did, will be much more manageable.

5.1 Future work

There is a lot of work to be done; our immediate future attention will be the item (1) below to expand our distributed testing environment followed by other proposed items:

1. Various JVMs in Linux and MacOS X distributed testing environments and clusters
2. Comparative study for Web Services-based implementation
3. Long-running distributed computation processes (e.g. MARF pattern recognition pipeline with very large data set over GIPSY)
4. Expand the architecture to mobile Java platforms

5.2 Acknowledgment

This work in part is supported by NSERC and the Faculty of Engineering and Computer Science, Concordia University, Montreal, QC, Canada. We thank reviewers for their constructive reviews and feedback.

References

- [1] J. Allard, V. Chinta, S. Gundala, and G. G. Richard III. JINI meets UPnP: An architecture for JINI/UPnP interoperability. In *Proceedings of the 2003 International Symposium on Applications and the Internet 2003*. SAINT, 2003.
- [2] Apache River Community. Apache River. [online], 2010. <http://incubator.apache.org/river/index.html>.
- [3] Edward A. Ashcroft, Anthony A. Faustini, Rangaswamy Jagannathan, and William W. Wadge. *Multidimensional Programming*. Oxford University Press, London, February 1995. ISBN: 978-0195075977.
- [4] S. Chen and P. Greenfield. QoS evaluation of JMS: An empirical approach. In *Proceedings of the 37th Hawaii International Conference on System Sciences*, 2004.
- [5] Eclipse contributors et al. Eclipse Platform. eclipse.org, 2000–2011. <http://www.eclipse.org>, last viewed February 2010.
- [6] R. Eggen and M. Eggen. Efficiency of distributed parallel processing using Java RMI, sockets, and CORBA. In *Proceedings of the 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'01)*. PDPTA, June 2001.
- [7] Bin Han. Towards a multi-tier runtime system for GIPSY. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2010.
- [8] M. Hapner, R. Burrige, R. Sharma, J. Fialli, and K. Stout. *Java(TM) Message Service API Tutorial and Reference*. Prentice Hall PTR, 2002. ISBN 0201784726.
- [9] Raganswamy Jagannathan and Chris Dodd. GLU programmer's guide. Technical report, SRI International, Menlo Park, California, 1996.
- [10] Raganswamy Jagannathan, Chris Dodd, and Iskender Agi. GLU: A high-level system for granular data-parallel programming. In *Concurrency: Practice and Experience*, volume 1, pages 63–83, 1997.
- [11] Yi Ji. Scalability evaluation of the GIPSY runtime system. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, March 2011.
- [12] Jini Community. Jini network technology. [online], September 2007. <http://java.sun.com/developer/products/jini/index.jsp>.
- [13] R. Joshi. A comparison and mapping of Data Distribution Service (DDS) and Java Message Service (JMS). Real-Time Innovations, Inc., 2006.
- [14] Serguei A. Mokhov. GIPSY: CVS service on Newton, a crash guide, June 2003. http://newton.cs.concordia.ca/~gipsy/cgi-bin/viewcvs.cgi/*checkout*/resources/doc/presentations/cvs.pdf?rev=HEAD.
- [15] Serguei A. Mokhov. Towards hybrid intensional programming with JLucid, Objective Lucid, and General Imperative Compiler Framework in the GIPSY. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, October 2005. ISBN 0494102934; online at <http://arxiv.org/abs/0907.2640>.
- [16] Serguei A. Mokhov. Towards syntax and semantics of hierarchical contexts in multimedia processing applications using MARFL. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 1288–1294, Turku, Finland, July 2008. IEEE Computer Society.
- [17] Serguei A. Mokhov, Joey Paquet, and Mourad Debbabi. Formally specifying operational semantics and language constructs of Forensic Lucid. In Oliver Göbel, Sandra Frings, Detlef Günther, Jens Nedon, and Dirk Schadt, editors, *Proceedings of the IT Incident Management and IT Forensics (IMF'08)*, LNI140, pages 197–216. GI, September 2008.
- [18] Serguei A. Mokhov, Joey Paquet, and Xin Tong. A type system for hybrid intensional-imperative programming support in GIPSY. In *Proceedings of C3S2E'09*, pages 101–107, New York, NY, USA, May 2009. ACM.
- [19] Nikolaos S. Papaspyrou and Ioannis T. Kassios. GLU# embedded in C++: a marriage between multidimensional and object-oriented programming. *Softw., Pract. Exper.*, 34(7):609–630, 2004.

- [20] Joey Paquet. Distributed eductive execution of hybrid intensional programs. In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC'09)*, pages 218–224, Seattle, Washington, USA, July 2009. IEEE Computer Society.
- [21] Joey Paquet. A multi-tier architecture for the distributed eductive execution of hybrid intensional programs. Unpublished, 2009.
- [22] Amir Hossein Pourteymour, Emil Vassev, and Joey Paquet. Towards a new demand-driven message-oriented middleware in GIPSY. In *Proceedings of PDPTA 2007*, pages 91–97, Las Vegas, USA, June 2007. PDPTA, CSREA Press.
- [23] Amir Hossein Pourteymour, Emil Vassev, and Joey Paquet. Design and implementation of demand migration systems in GIPSY. In *Proceedings of PDPTA 2009*. CSREA Press, June 2008.
- [24] Amir Hossein Pouteymour. Comparative study of Demand Migration Framework implementation using JMS and Jini. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, September 2008.
- [25] Sun Microsystems, Inc. Java Message Service (JMS). [online], September 2007. <http://java.sun.com/products/jms/>.
- [26] The GIPSY Research and Development Group. The General Intensional Programming System (GIPSY) project. Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2002–2011. <http://newton.cs.concordia.ca/~gipsy/>, last viewed February 2010.
- [27] Xin Tong. Design and implementation of context calculus in the GIPSY. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, April 2008.
- [28] Emil Vassev and Joey Paquet. A general architecture for demand migration in a demand-driven execution engine in a heterogeneous and distributed environment. In *Proceedings of the 3rd Annual Communication Networks and Services Research Conference (CNSR 2005)*, pages 176–182. IEEE Computer Society, May 2005.
- [29] Emil Vassev and Joey Paquet. A generic framework for migrating demands in the GIPSY's demand-driven execution engine. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*, pages 29–35. CSREA Press, June 2005.
- [30] Emil Iordanov Vassev. General architecture for demand migration in the GIPSY demand-driven execution engine. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, June 2005. ISBN 0494102969.
- [31] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, London, 1985.
- [32] Kaiyu Wan. *Lucx: Lucid Enriched with Context*. PhD thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2006.
- [33] Ai Hua Wu. *OO-IP Hybrid Language Design and a Framework Approach to the GIPC*. PhD thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2009.

A How to run the Jini DMS and the JMS DMS in the GIPSY project

This documents introduces the steps to deploy and start the GIPSY runtime system in 32-bit Windows.

1. check out the GIPSY project [14]. The project is an Eclipse project so you can simply imported into Eclipse IDE for Java Developers [5]. However, before you import it into Eclipse, remember to stop Eclipse from cleaning the bin folder as all the DMS executables are there. For example, in the case of Eclipse Helios, once the IDE is open, go to Windows-¿Preferences-¿Java-¿Compiler-¿Building, and uncheck the option “Scrub output folders

when cleaning projects”, and go back to the menu bar, click Project, and uncheck the option “Build Automatically” so that you will need to build the project manually. Then you can import the GIPSY project and build it manually.

2. Once the GIPSY project is built, you can go to the project’s home folder, then go to the directory
 - bin
 - multitier
 - .
3. If you want to start Jini DMS, open StartGMTNode.config, and set the value of the property gipsy.GEE.multitier.Node.DSTConfigs as ../jini/DST.config
4. If you want to start JMS DMS, open StartGMTNode.config, and set the value of the property gipsy.GEE.multitier.Node.DSTConfigs as ../jms/DST.config
5. Double click startGMTNode.bat.

B How to run the Jini DMS in the GIPSY project

This document introduces the steps to compile and run the Jini code in the GIPSY project in Windows XP. To use this guide, readers are required to have basic Java and Eclipse experience, basic Jini knowledge (for example, service, lookup, JavaSpace, etc), and the basic understanding of the GIPSY project structure.

1. Get and install the appropriate software, and import the GIPSY project [14]. NOTE: If the software below cannot be found online, please check the GIPSY tools repository
 - (a) JDK 6 update 14 or later (<http://java.sun.com/javase/downloads/index.jsp>). It is better to set the JAVA_HOME environment variable.
 - (b) Eclipse IDE for Java Developers [5]. Unpack the IDE, open it and import the GIPSY project from the GIPSY CVS into the Eclipse.
 - (c) Jini Technology Starter Kit v2.1 [12]. The installation should require no administrator accounts. The term JINI_HOME would be used in this manual to refer the installation directory.
2. Compile the Jini code of the GIPSY project NOTE: Point 1 and 2 should be done already in the GIPSY project when you get it.
 - (a) Copy the jini-core.jar and jini-ext.jar from JINI_HOME/lib into the gipsy/lib
 - (b) Configure the project Build Path by adding the above jars inside lib through “Add JARs”
 - (c) Build the project automatically or manually.
3. Start the Jini service
 - (a) Go to JINI_HOME/installverify, and double-click the Launch All shortcut. The shortcut should launch a service window and a Service Browser window.

- (b) In the Service Browser window, click the “Registrar”, and select the registrar representing your computer. Then there would be 6 services appear in the “Matching Services” area, including JavaSpace05, LookupDiscoveryService, ServiceRegistrar and TransactionManager.
 - (c) Leave the two windows open and do not touch them unless you want to shut down all the services.
4. Run the code requiring only JavaSpace. NOTE: Currently there are two groups of Jini scenarios. The first one consists of `DemandDispatcher`, `DemandDispatcherClient` and the `DemandDispatcherAgent` under the `gipsy.GEE.IDP` package. The second scenario consists of the `gipsy.GEE.IDP.DemandDispatcher`, and the DGT class in the `gipsy.GEE.IDP.DemandGenerator.simulator.jini` package, and the `Worker` class in the `gipsy.GEE.IDP.DemandGenerator.simulator.jini` package.
- (a) Open Eclipse and run the classes within the same groups mentioned above with the `main()` method.
5. Run the code requiring both JavaSpace and RMI
- (a) Use command window to go to the `gipsy/bin` directory.
 - (b) Open the `startJiniHTTPServer.bat` in edit mode, and check if all the paths are correct.
 - (c) Double-click the `startJiniHTTPServer.bat`.
 - (d) Double-click the `startJiniRMID.bat`.
 - (e) Open Eclipse and run the `gipsy.GEE.IDP.DemandGenerator.jini.rmi.JINITransportAgent`. and the `gipsy.GEE.IDP.DemandGenerator.simulator.DGT` in the `gipsy.GEE.IDP.DemandGenerator.simulator` package.

Note:

Please make sure that the `startJiniHTTPServer.bat` and the `startJiniRMID.bat` are in the `gipsy/bin` folder. If they are missing, please refer to the following content. Please make sure the settings in the content are consistent with your own JRE directories as shown in Figure 8 and Figure 9.

```
set RUNTIME_JAR="D:\Program Files\Java\jre6\lib\rt.jar"
set JINIHOME_BACKSLASH="D:\Program Files\jini2_1"
set JINI_CLASSPATH=.;%RUNTIME_JAR%;%JINIHOME_BACKSLASH%\lib\jini-core.jar;%JINIHOME_BACKSLASH%\lib\jini-ext.jar;%JINIHOME_BACKSLASH%\lib\reggie.jar;%JINIHOME_BACKSLASH%\lib-dl\reggie-dl.jar;%JINIHOME_BACKSLASH%\lib\mahalo.jar;%JINIHOME_BACKSLASH%\lib-dl\mahalo-dl.jar;%JINIHOME_BACKSLASH%\lib\outrigger.jar;%JINIHOME_BACKSLASH%\lib-dl\outrigger-dl.jar;%JINIHOME_BACKSLASH%\lib\tools.jar;%JINIHOME_BACKSLASH%\lib\sun-util.jar;

java -jar -classpath %JINI_CLASSPATH% %JINIHOME_BACKSLASH%\lib\tools.jar -port 8085 -dir . -verbose
```

Figure 8: `startHTTPServer.bat`

```
rmdir -J-Djava.security.policy=gipsy/GEE/IDP/config/jini.policy
```

Figure 9: `startJiniRMID.bat`

Index

API

- DemandDispatcher, 15
- DemandDispatcherAgent, 15
- DemandDispatcherClient, 15
- DGT, 15
- gipsy.GEE.IDP, 15
- gipsy.GEE.IDP.DemandDispatcher, 15
- gipsy.GEE.IDP.DemandGenerator.jini.rmi.JINITransportAgent,
15
- gipsy.GEE.IDP.DemandGenerator.simulator,
15
- gipsy.GEE.IDP.DemandGenerator.simulator.jini,
15
- gipsy.GEE.IDP.DemandGenerator.simulator.jini.WorkerJTA,
15
- JAVA_HOME, 14
- JINI_HOME, 14
- main(), 15
- Worker, 15

C++, 1

DMF, 2

DMS, 1

Files

- gipsy/bin, 15
- gipsy/lib, 14
- jini-core.jar, 14
- jini-ext.jar, 14
- JINI_HOME/installverify, 14
- JINI_HOME/lib, 14
- startJiniHTTPServer.bat, 15
- startJiniRMID.bat, 15

Forensic Lucid, 1

Frameworks

- DMF, 2

GIPSY, 1–4, 9, 13, 14

Jini, 1–7, 9–11, 13, 14

JLucid, 1

JMS, 1–7, 9–11, 13, 14

JOOIP, 1

Lucid, 1

Tensor Lucid, 1